NAVAL POSTGRADUATE SCHOOL
Monterey, California

THESIS

A CONCEPTUAL LEVEL DESIGN OF A DESIGN DATABASE FOR THE COMPUTER-AIDED PROTOTYPING SYSTEM

by

Bryant Steven Douglas

March 1989

Thesis Advisor: Valdis Berzins

REPORT DOCUMENTATION PAGE

1a REPORT SECURITY CLASSIFICATION UNCLASSIFIED	1b RESTRICTIVE MARKINGS
2a SECURITY CLASSIFICATION AUTHORITY	3 DISTRIBUTION / AVAILABILITY OF REPORT Approved for public release; Distribution is unlimited
2b DECLASSIFICATION / DOWNGRADING SCHEDULE	
4 PERFORMING ORGANIZATION REPORT NUMBER(S)	5 MONITORING ORGANIZATION REPORT NUMBER(S)
6a NAME OF PERFORMING ORGANIZATION Naval Postgraduate School 6b OFFICE SYMBOL (If applicable) 52	7a NAME OF MONITORING ORGANIZATION Naval Postgraduate School
6c ADDRESS (City, State, and ZIP Code) Monterey, CA 93943-5000	7b ADDRESS (City, State, and ZIP Code) Monterey, CA 93943-5000
8a NAME OF FUNDING / SPONSORING ORGANIZATION 8b OFFICE SYMBOL (If applicable)	9 PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER
8c ADDRESS (City, State, and ZIP Code)	10 SOURCE OF FUNDING NUMBERS PROGRAM ELEMENT NO / PROJECT NO / TASK NO / WORK UNIT ACCESSION NO

11 TITLE (Include Security Classification)
A CONCEPTUAL LEVEL DESIGN OF A DESIGN DATABASE FOR THE COMPUTER-AIDED PROTOTYPING SYSTEM

12 PERSONAL AUTHOR(S) Douglas, Bryant S. [illegible]

13a TYPE OF REPORT Master's Thesis	13b TIME COVERED FROM ______ TO ______	14 DATE OF REPORT (Year, Month, Day) 1989 March	15 PAGE COUNT 98

16 SUPPLEMENTARY NOTATION The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Goverment

17 COSATI CODES

FIELD	GROUP	SUB-GROUP

18 SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
Object-oriented design; Object-oriented database; Engineering database; Computer Aided Software Engineering (CASE); Rapid Prototyping; Design database, Theses. (edc

19 ABSTRACT (Continue on reverse if necessary and identify by block number)
Vast amounts of evolving data are created in the design of hard real-time software systems. This data must be managed so that it can be stored and retrieved according to the needs of design engineers. In the Computer-Aided Prototyping System (CAPS), a Design Database (DDB) must manage the storage and retrieval of the entire Prototype System Description Language (PSDL) program. This thesis presents a conceptual design and initial implementation of a Design Database (DDB) for the Computer-Aided Prototyping System (CAPS).

20 DISTRIBUTION / AVAILABILITY OF ABSTRACT [✓ UNCLASSIFIED/UNLIMITED] ☐ SAME AS RPT ☐ DTIC USERS	21 ABSTRACT SECURITY CLASSIFICATION Unclassified
22a NAME OF RESPONSIBLE INDIVIDUAL Valdis Berzins	22b TELEPHONE (Include Area Code) (408) 646-2461 22c OFFICE SYMBOL 52Be

A CONCEPTUAL LEVEL DESIGN OF A DESIGN DATABASE FOR THE COMPUTER-AIDED PROTOTYPING SYSTEM

by

Bryant S. Douglas
Lieutenant, United States Navy
B.S.B.A., University of Missouri - Columbia, 1983

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN INFORMATION SYSTEMS

from the

NAVAL POSTGRADUATE SCHOOL
March 1989

Author: Bryant S Douglas
Bryant S. Douglas

Approved by: Valdis Berzins
Valdis Berzins, Thesis Advisor

T. Hamid
Tarek Abdel-Hamid, Second Reader

[illegible]
David R. Whipple, Chairman
Department of Administrative Sciences

K.T. Marshall
Kneale T. Marshall
Dean of Information and Policy Sciences

ABSTRACT

Vast amounts of evolving data are created in the design of hard real-time software systems. This data must be managed so that it can be stored and retrieved according to the needs of design engineers. In the Computer - Aided Prototyping System (CAPS), a Design Database (DDB) must manage the storage and retrieval of the entire Prototype System Description Language (PSDL) program. This thesis presents a conceptual design and initial implementation of a Design Database (DDB) for the Computer - Aided Prototyping System (CAPS).

[illegible]

THESIS DISCLAIMER

TABLE OF CONTENTS

LIST OF FIGURES

ACKNOWLEDGMENT

This thesis is dedicated in loving memory to my father, W.B. Douglas. I would not be here today were it not for his love and guidance. I would like to thank my mother, Frances Douglas, for her encouragement and words of kindness throughout my life.

I would like to thank Professor Valdis Berzins for taking on a thesis student who did not possess the usual background required for this type of thesis. I would like thank Professor Luqi for her encouragement when I was discouraged and frustrated.

Last but not certainly not least, I would like to thank my loving wife and "real" thesis advisor, Carla, for her persistence and her love throughout this trying time.

I. INTRODUCTION

A. BACKGROUND

The development of hard real-time and embedded software systems is an extremely complex and expensive process. A software development methodology which will reduce the development costs, increase the productivity rates, and reduce the maintenance costs of these systems is long overdue. The prevailing ideas of today are computer-aided rapid prototyping, software reusability, and the use of an executable high-level specification language. The goal of the Computer-Aided Prototyping System (CAPS) is to integrate all of these ideas and more, into one software development tool. [Ref. 1:p. 66]

In 1985, the United States Department of Defense (DOD) spent roughly $11 billion dollars in software costs and is estimated to spend $36 billion in 1995 [Ref. 2:p. 43]. The majority of these costs are involved in the development and maintenance of embedded systems [Ref. 3:p. 13]. These costs certainly inspire one to think of software development in terms of a crisis. As stated by Booch:

> The symptoms of the software crisis appear in the form of software that is nonresponsive to user needs, unreliable, excessively expensive, untimely, inflexible, difficult to maintain, and not reusable [Ref. 3:p. 2].

Thus, a software development tool which can provide a 20 percent improvement in software productivity will save the DOD and the American taxpayer billions of dollars.

1. Hard Real-Time and Embedded Software Systems

The development of hard real-time and embedded software systems creates additional problems in the software development process. As pointed out by Booch:

they all generally share a set of common characteristics, namely:

- Large. Thousands/millions of lines of code.
- Long-lived. 10 to 15 years.
- Continuous Change. Due to changing requirements.
- Physical constraints. In target hardware address space/speed.
- High reliability. Also fault-tolerant. [Ref. 3:p. 13]

Each of these characteristics makes embedded systems difficult to develop. For this reason, the DOD mandated the use of Ada in all embedded computer software programs, whether new programs or upgrades to existing ones [Ref. 4:p. 33]. The effects of this decision on software development costs may not be felt immediately , but the long term savings of a universal programming language for embedded systems will be realized.

A hard real-time system is defined as a software or firmware controlled system that performs all its process functions within critical specified time constraints [Ref. 5:p. 3].

An embedded system, on the other hand, is one that is part of a larger system [Ref. 3:p. 3]. However, embedded systems usually require hard real-time constraints. A real-time system is more difficult to develop than a non-real-time system. As discussed in Ref. 5, some of the difficulties include:

- Handling of stringent time requirements and performance specifications.
- Interfacing with a real-time clock.
- Control of hardware devices such as communication lines, terminals, and resources.
- Processing of messages that arrive at irregular intervals, with fluctuating input rates, and with different priorities.
- Control of fault conditions with facilities for various degrees of recovery.
- Handling of queues and buffers for storage of messages and data items.
- Modeling of concurrent conditions into a proper set of concurrent processes.
- Allocation and control of concurrent processes to processors.
- Handling of communication and synchronization between concurrent processes.
- Protection of data shared between concurrent processes.
- Scheduling and dispatching (including priority handling) of concurrent processes.

Due to these demanding requirements, the development of hard real-time and embedded systems is an expensive and time consuming process. A development methodology which can reduce the development cost and time of this process will be of great benefit to the DOD.

2. The Computer-Aided Prototyping System

The Computer-Aided Prototyping System is one attempt to improve the productivity and reliability of software through the use of computer-aided rapid prototyping via specification and reusable components [Ref. 1:p. 66]. This approach to rapid prototyping uses a specification language called Prototype System Development Language (PSDL) integrated with a set of software tools [Ref. 1:p. 66]. Figure 1 gives a graphical representation of CAPS [Ref. 6:p. 8]. The major components of CAPS are a user interface consisting of a syntax directed editor and graphical editor, a design management system consisting of a software base management system and design database, and an execution support system consisting of a translator, dynamic scheduler, and debugger. For further explanations of the above systems see Refs. 6, 7, 8, 9, 10, 11, and 12.

3. The Design Database

Vast amounts of evolving data are created in the design of hard real-time software systems. Conventional database management systems (DBMS) were designed for business applications and as such are insufficient to handle the needs of computer-aided design (CAD) applications. The data must be managed so that it can be stored and retrieved according to the needs of design engineers. In CAPS, the Design Database (DDB) must manage the storage and retrieval of the Prototype System Description Language (PSDL) program. The DDB must be a specialized DBMS which will store PSDL specifications in a hierarchical format.

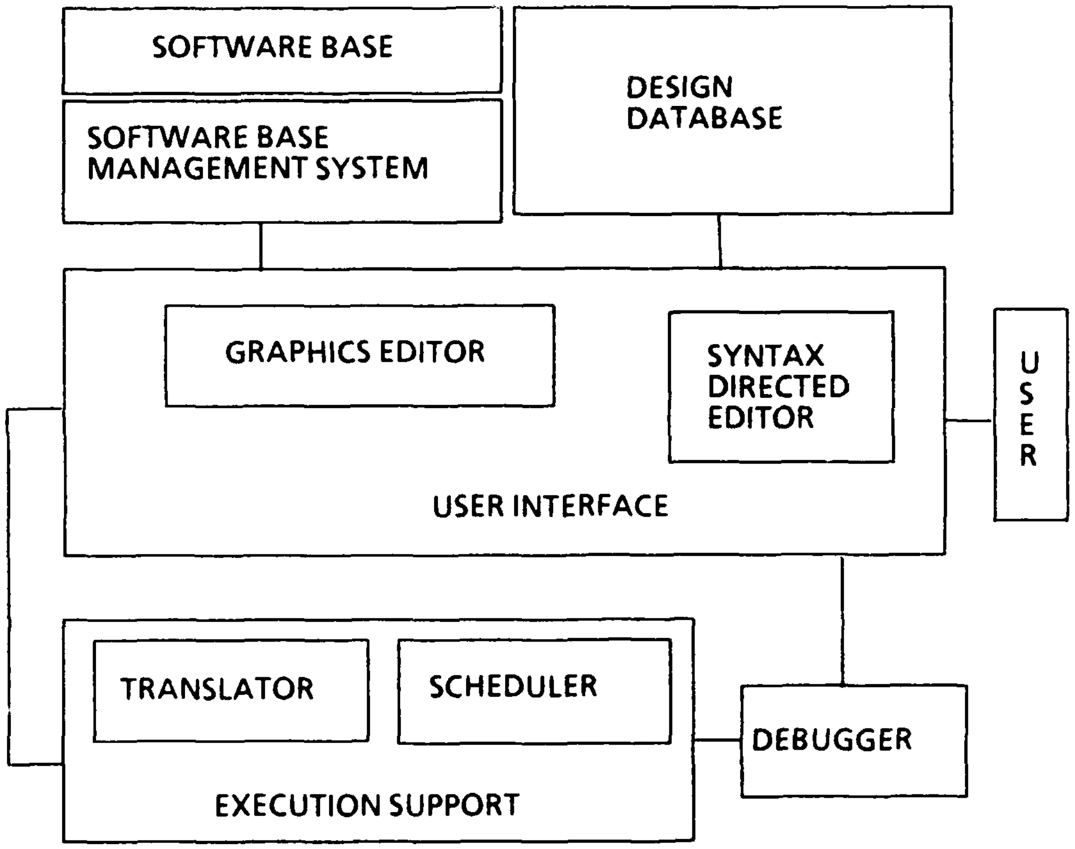

Figure 1. Computer-Aided Prototyping System

4. The Object-Oriented Approach

As stated by Ketabchi:

> There is an increasing interest in developing object-oriented database management systems to manage the large amount of data involved in computer-aided design (CAD) applications [Ref. 13:p.44].

In the past three years, several object-oriented database management systems have emerged. The impact of these systems on the software development process are just beginning to be felt. The object-oriented approach represents a true paradigm shift [Ref. 14:p. 386].

B. OBJECTIVES

The objective of this thesis is the development of a conceptual level design and initial implementation for the Design Database of the Computer-Aided Prototyping System. This design will be the basis for further research leading to full implementation of the Design Database.

C. ORGANIZATION

Chapter II contains a survey of recent work in the area of software engineering databases with an emphasis on the object-oriented approach. An introduction to Vbase, a state of the art object-oriented DBMS will conclude Chapter II. Chapter III contains the actual design of the Design Database. Chapter IV shows the feasibility of the Design Database using Vbase. Conclusions and recommendations will be presented in Chapter V.

II. ENGINEERING DATABASES AND THE OBJECT-ORIENTED APPROACH

A. ENGINEERING DATABASES

There have been various attempts at achieving a solution to the management of design databases. According to Berzins and Ketabchi:

> Four different approaches to the solution exist:
>
> (1) Developing a new DBMS, called a design DBMS (DDBMS), equipped with facilities required in design applications.
>
> (2) Enhancing the current DBMSs by adding new capabilities.
>
> (3) Building a layer of software on top of current DBMSs to compensate for their deficiencies.
>
> (4) Using a special-purpose file manager that views the DBMS as an application. [Ref. 15:p. 94]

Sherpa Data Management System provides an example of the first approach [Ref. 16:p. 55]. Starburst, Almaden, and Postgres provide examples of the second approach [Ref 16:p. 57]. The third approach is the most popular approach in the industry, because it allows the use of off-the-shelf DBMS and can be made rapidly operational [Ref. 15:p. 94]. With the invention of object-oriented technology, the first approach will become the most popular because it also allows the use of off-the-shelf DBMS but with enhanced capabilities.

B. THE OBJECT-ORIENTED APPROACH

An object-oriented database management system (OODBMS) must provide persistence, concurrency, recovery, transaction management, authorization, and security [Ref. 16:p. 60]. An OODBMS is an object-oriented system and as such it must also provide the following capabilities:

- Objects
- Active data
- Abstraction
- Extensibility [Ref. 16:p. 61]

An OODBMS should provide application-oriented capabilities such as:

- Version and configuration control for CAD applications.
- Dynamic creation of classes.
- Recursive classes, multiple inheritance, and extensive tool interface capabilities.
- Support for multimedia objects, distributed environments, and graphics. [Ref. 16:p. 62]

An object-oriented DBMS is one that supports persistency, values, an extensible set of data structures, an extensible set of operations, and abstractions [Ref. 16:p. 76].

C. VBASE

The Vbase object-oriented database management system is a product of Ontologic, Inc. As Ontologic puts it:

> Vbase is an object database system, providing an integrated software development platform which combines the latest advances in compiler design and database management techniques [Ref. 17:p. 1].
>
> The Vbase Integrated Object System is a database and language platform for rapidly and inexpensively building sophisticated commercial and engineering applications [Ref. 17:p. 5].

The Vbase system environment consists of the following components:

- Vbase Database. Persistent objects are stored in the Vbase Database.
- Object Language. Type Definition Language (TDL) is the Vbase data definition language. It is used to define object types in the database. "C" Object Processor (COP) is the Vbase data manipulation language. It is an object extension of Kernighan and Ritchie standard C. It is used to implement the operations of the object types defined using TDL.
- System Type Library. The system type library contains many object types which provide powerful building blocks for the application developer.
- Integrated Tool Set (ITS). A single tool combining the functionality of a source browser, database browser, and a source debugger.
- Object SQL. The Vbase implementation of the SQL standard query language. [Ref. 17:p. 6]

Figure 2 presents a graphical representation of Vbase.

The steps involved in a typical Vbase database design are listed below:

- Identify the objects.
- Identify their properties as much as possible.
- Identify the frequent operations performed on the objects.
- Define the objects using TDL.
- Compile and debug the TDL definition of the objects.
- Develop COP routines to implement the operations.
- Compile and debug the COP programs.
- Develop C or COP user applications. [Ref. 16:p. 8]

Vbase is a powerful tool for implementing and maintaining large software applications. Its integration of compiler technology with database functionality in a strongly-typed system provides both sophisticated modeling and efficient language and database performance.

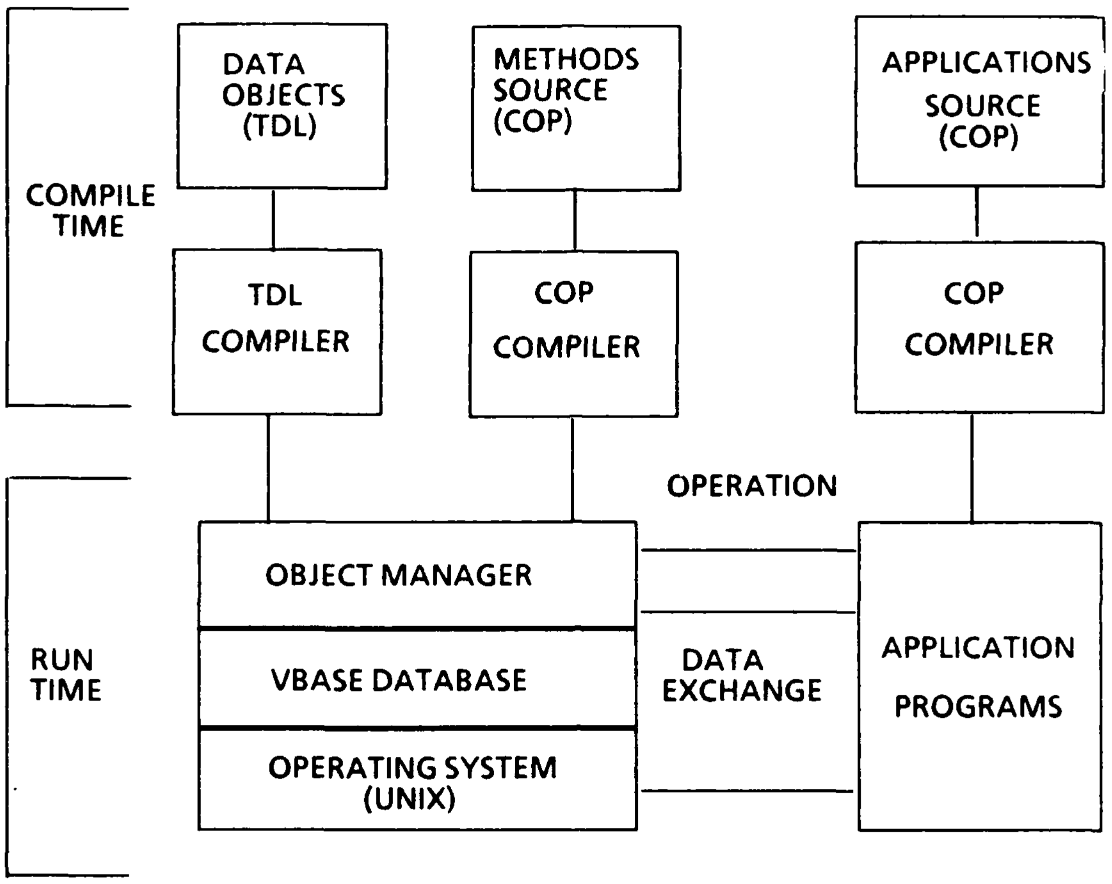

Figure 2. Vbase Overview

III. CONCEPTUAL DESIGN FOR THE DESIGN DATABASE

A. REQUIREMENTS FOR THE DESIGN DATABASE IN RAPID PROTOTYPING

The purpose of the Design Database (DDB) in the Computer-Aided Prototyping System (CAPS) is to manage the project database so that it can be stored and retrieved according to the needs of design engineers. The requirements analysis was conducted using the specification language SPEC [Ref. 18]. SPEC is a language for giving black-box specifications in the early stages of software design [Ref. 18:p. 1].

As stated by Berzins:

> The goal of requirements analysis is to establish the purpose of the proposed software system and to establish constraints and boundary conditions on the rest of the software development process [Ref. 19:p. 1-2].

The result of the requirements analysis should include the following:

- A model of the system's environment.
- A description of the goals of the system and the functions it must perform.
- Performance constraints on the system.
- Constraints on the implementation of the system.
- Resource constraints for the development project.
- A specification of the external interfaces of the system. [Ref. 19:p. 2-1]

1. Environment Model

As stated by Berzins:

> The environment model must supply the concepts needed for describing the world in which the proposed system will operate. These concepts consist of the types of objects in that world, the attributes of those objects, the relations between those objects, and the laws governing those objects and relations. [Ref. 19:p. 2-2]

The environment model for the Design Database is shown below. The model was formulated in terms of reusable model components. A reusable component is a definition of a general type or relation [Ref. 19:p. 2-8]. The reusable components used are shown in Appendix A [Ref. 19:p. 2-9]. The model is expressed in a simple notation that is explained as it appears. Explanatory comments are preceded by a "--" symbol. A grammar for the notation is given in Appendix B [Ref. 19:p. 2-21].

```
type CAPS

 a_kind_of(software_system,CAPS)

  -- The Computer-Aided Prototyping System (CAPS) is a
  -- software_system.

type psdl

 a_kind_of(psdl,language)

  -- PSDL is a language for expressing specifications.

 created_by(every psdl,a user_interface)

  -- All specifications are created in the user interface.
```

type design_database

a_kind_of(design_database,software_system)

-- The design_database is going to be a software_system.

part_of(a design_database,every CAPS)

-- The design_database is part of CAPS

unique(design_database)

-- There will be only one instance of the design_database for each
-- project.

proposed(a design_database)

-- I am going to build a design_database.

controls(a design_database,node)

-- The design_database controls the design data by
-- managing collections of data called nodes.

type design_engineer

a_kind_of(design_engineer,user)

-- The design engineer is a user of the system.

uses(every design_engineer,a user_interface)

-- The model includes only the design engineers that will interact
-- with the design_database via the user_interface.

type user_interface

a_kind_of(user_interface, software_system)

-- The user_interface is a software_system.

part_of(a user_interface,every CAPS)

-- The user_interface is a part of CAPS.

created_by(a user_interface,every node)

-- The user_interface is the only source for data.

type data

-- any kind of data that is used by a software system

uses(a software_system, every data)

relation reads(software_system, data)

-- true if the data is an input for the software system

reads(any software_system, any data) => uses (software_system,data)

relation writes(software_system, data)

-- true if the data is an output for the software system

writes(any software_system, any data) => uses (software_system,data)

relation updates (software_system, data)

-- true if the data is both an input and an output for the system

updates(any software_system, any data)

<=> reads(software_system, data) & writes(software_system, data)

type node

a_kind_of(node,data)

-- A node is the basic structure for maintaining the design database.

needed_for(node,every specification)

-- Every specification created in the system will be stored in a node.

created_by(every node,a user_interface)

-- All nodes are created via the user interface.

-- The attributes of a node are listed below.

specification(node):psdl

implementation(node):psdl

control_constraints(node):psdl

graphic_record(node):graphic_record

-- A node consists of a specification, a graphic record,
-- an implementation, and control constraints.

text(node):psdl_file

type graphic_record

a_kind_of(graphic_record,data)

part_of(node,graphic_record)

-- A graphic record is one input into in a node.

created_by(a user_interface,every graphic_record)

type implementation

a_kind_of(implementation,data)

part_of(node,implementation)

-- An implementation is one input into a node.

created_by(a user_interface,every implementation)

type control_constraints

a_kind_of(control_constraints,data)

part_of(node,control_constraints)

-- Control constraints are part of the data in a node.

created_by(a user_interface,every control_constraints)

type psdl_file

a_kind_of(psdl_file, data)

-- A file containing the PSDL program will be the ultimate output.

2. High Level Goals

The requirements for the DDB are formalized by writing a description of the goals of the system and the functions it must perform in terms of the model. A major part of the requirements analysis is turning the informal problem statement into a precise, testable, and feasible set of requirements. The high level goals for the Design Database are shown below.

G1: The purpose of the system is to store the levels of a PSDL design in a hierarchical format.

G1.1: The system must allow design engineers to retrieve the levels for review or editing.

G1.2: The system must be able to create and insert new levels into the structure.

G1.2.1: The system must interface with the user interface for inputs to the database.

G2: The system must be able to generate the entire PSDL program.

3. Constraints

There are three types of constraints for a software system: implementation, performance, and resource. The constraints for the Design Database are given below.

Implementation constraints:

C1: The design database must be implemented with a DBMS which is compatible with the Sun workstation and Unix operating system.

Performance constraints:

C2: The responses of the design database must be fast enough not to irritate the design engineers using the system.

Resource constraints:

C3: The Design Database will be developed by thesis students at the Naval Postgraduate School.

B. CONCEPTUAL MODEL OF THE DESIGN DATABASE

The DDB is a hierarchical storage structure for the PSDL program. This structure is a tree consisting of atomic and composite nodes. Figure 3 illustrates this structure. Both types of nodes contain a node name, a PSDL specification, an implementation, and parent <-> child relationship information. An atomic node's implementation is ADA code. A composite node's implementation is graph. A graphic implementation may contain timing constraints in the form of control constraints. Each level of the tree is created by the decomposition of the parent node. The decomposition process is complete when all leaf nodes are atomic. Figures 4 and 5 present graphical representations of atomic and composite nodes.

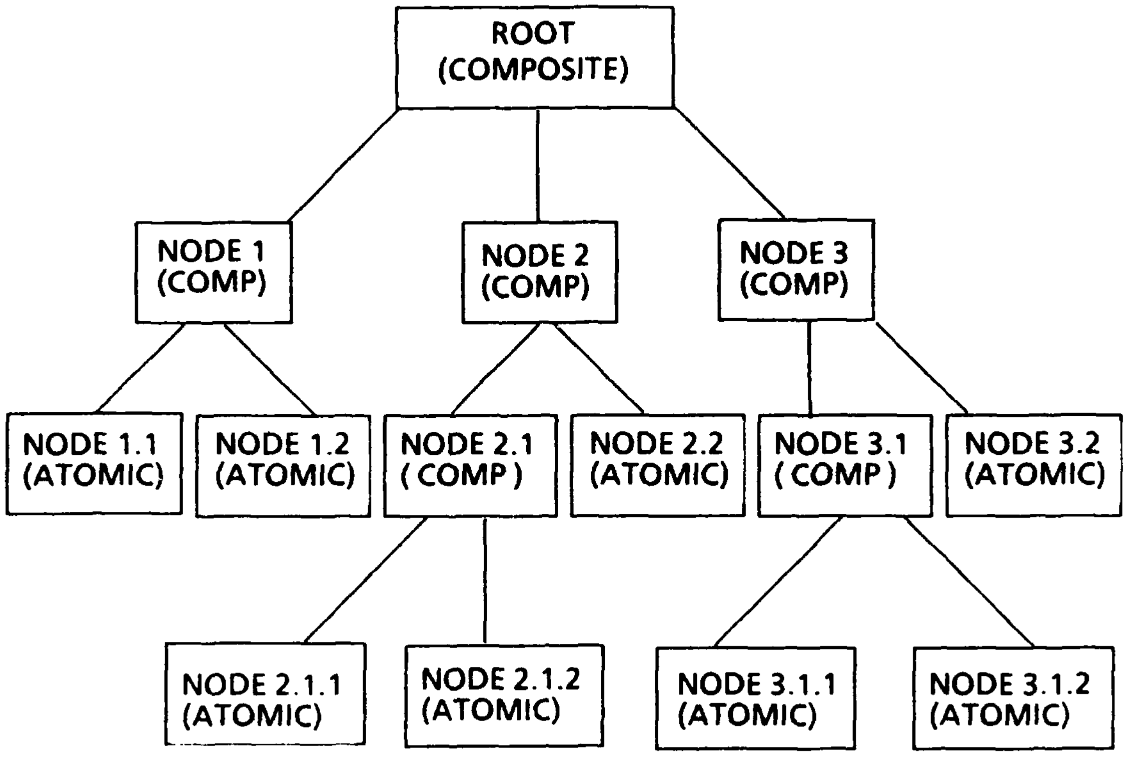

Figure 3. Conceptual DDB Structure

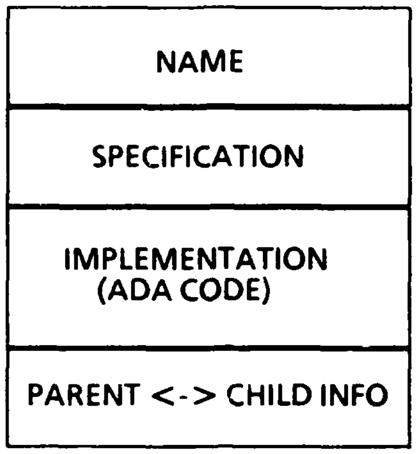

Figure 4. Atomic Node

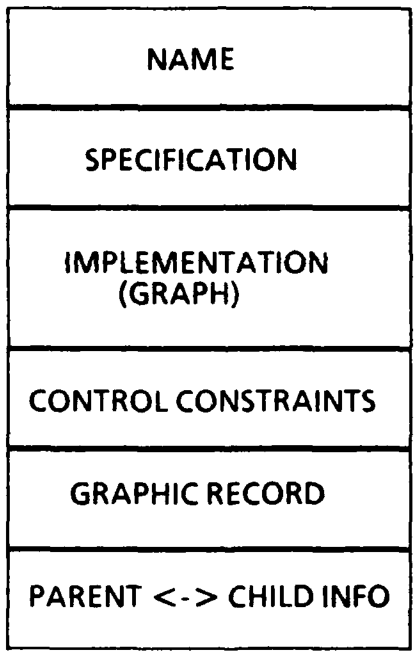

Figure 5. Composite Node

C. FUNCTIONAL SPECIFICATIONS FOR THE DESIGN DATABASE

As described by Berzins:

A functional specification is a precise black-box model of the proposed software system. The result of the functional specification phase is an event model of the system to be built, expressed in a Spec language. In the event model, computations are described in terms of modules, events, and messages. A module is a black box that interacts with other modules only by sending and receiving messages. An event occurs when a message is received by a module at a particular instant of time. A message is a data packet that is sent from one module to another. [Ref. 18:p. 2]

A further description of the Spec language by Berzins states:

The Spec language provides a means for specifying the behavior of three different types of modules: functions, state machines, and abstract data types. Function modules are immutable, and calculate functions on data types. A machine is a module with an internal state. An abstract data type consists of a set of instances and a set of primitive operations involving instances. [Ref. 18:p. 3]

The functional specifications begin with a skeleton of a specification with places for each of the missing details to be filled in later. The initial specifications are shown below.

```
MACHINE design_database

 -- The design database is a machine because it is time sensitive.

 INHERIT user_interface_interface

 -- The system will interface with the user interface portion of the
 -- CAPS system therefore, inheritance relations exist.

 STATE

 INVARIANT true

 INITIALLY true

END
```

```
MACHINE user_interface

 -- The user_interface is an external system that sends messages to
 -- the design database.

 STATE

 INVARIANT true

 INITIALLY true

END
```

The next step is to make a list of the messages in each interface by consulting the requirements. The user interface is the only interface for the design database. The following messages are produced corresponding to the goals of the system:

user_interface_interface

create_root_node	G1, G1.2
create_child_node	G1, G1.2
delete_node	G1, G1.2
lookup_node	G1.1
get_parent	G1.1
get_child	G1.1
traverse_tree	G2

These messages have covered all of the goals in the goal tree with the exception of G1.2.1. This goal is an assumption about the environment.

The result of the user_interface interface messages are shown below.

MACHINE user_interface_interface

```
 -- The skeleton specification begins by identifying the interface
 -- messages.

 STATE (tree:map{node,set{node}})

 INVARIANT existing_nodes(node)

 INITIALLY node = {}


 MESSAGE lookup(node_name) -- G1.1

 -- Find the node requested.

  WHEN ? -- node found

   REPLY node -- return node contents

  OTHERWISE REPLY EXCEPTION node does not exist


 MESSAGE get_parent(node) -- G1.1

 -- Find the parent of the node requested.

  WHEN ? -- parent found

   REPLY node -- return parent node

  OTHERWISE REPLY EXCEPTION node is the root
```

MESSAGE get_child(node) -- G1.1

-- Find the child or children of the node requested.

REPLY set{node} -- return child node(s)

MESSAGE create_root_node(node) -- G1.1, G1.2

-- Create a new node and insert into the top of the hierarchy

WHEN? -- node created

REPLY done

TRANSITION? -- add root

MESSAGE create_child_node(node) -- G1.1, G1.2

-- Create a new node and insert into the hierarchy correctly

REPLY done

TRANSITION? -- add node

MESSAGE delete_node(node) -- G1.1, G1.2

-- Find the node and delete it and all children from the structure.

WHEN? -- node found

REPLY done

TRANSITION? -- remove node and children

OTHERWISE REPLY EXCEPTION node does not exist

```
MESSAGE traverse_tree(node) -- G2
-- Find the requested root node and display all children if they all
-- consist of PSDL specifications.
 WHEN? -- node found
  REPLY set{node}
 OTHERWISE REPLY EXCEPTION psdl program does not exist
END
```

The goal of functional specification is to construct a model of the proposed system as it is visible to the users [Ref. 19:p. 1-3]. The concepts that the users will be expected to know and the details of the interfaces are defined. The functional specification does not include any information on how the system behavior is to be realized. The result is a set of definitions for the system concepts and interfaces. The major functions of the DDB are:

- Store the levels of a PSDL program in a hierarchical format by specification.
- Retrieve the levels of a PSDL program in a hierarchical format by specification for review or editing.
- Create and insert new levels of a PSDL program in a hierarchical format by specification.
- Generate the entire PSDL program.

A brief example to illustrate the expected patterns of use of the methods provided by the DDB follows. To construct a prototype, the PSDL specification of the root operator is entered. At this point, the DDB would create a root node. Assuming the root node is composite, the node would be decomposed into children operators. The DDB would create child nodes for each decomposition. The decomposition process would continue until all leaf nodes are atomic. This process will be time consuming and the prototype will be complex. For this reason, the functions of retrieving nodes, parent-child relationships, and deleting nodes will be required. The tree will be traversed and the entire PSDL program produced once all leaf nodes are atomic.

D. ARCHITECTURAL DESIGN

The next step in the design process is to develop an architectural design.

> An architectural design is a model of the proposed system capturing the aspects of its behavior and structure relevant to the development team. The behavior of a system consists of its interactions with other systems, while the structure of a system consists of its component parts and their interconnections. [Ref. 19:p. 4-53]

The functional specification is a subset of the architectural design. The functional specification is the least detailed view of the system.

"The goal of the architectural design is to break up the proposed system into a set of small modules." [Ref. 19:p. 4-54] A module is defined as a self contained unit of code.

> A module is both a self-contained abstraction and a unit of work. Modules have several different views: black-box specification, parts lists, glass-box specifications, and programs. [Ref. 19:p. 4-54]

Black-box specifications are expressed in terms of the event model at the architectural design and functional specification stage. The parts lists contains the set of lower level modules used directly in the implementation. Glass-box specifications are represented by pseudo-code. Programs are produced in the implementation phase.

[Ref. 19:p. 4-54] The black-box specifications will be shown below. The parts lists and pseudo-code are not contained. Programs will be discussed in Chapter IV.

Black-box specifications

Type Node

```
Model (specification implementation graphic_record
        control_constraints: string)

-- The following messages are used to replace the current value of
-- a node's property with a new value.
```

MESSAGE add_graphic_record(node)

```
  TRANSITION? -- update node graphic record info
```

MESSAGE add_implementation(node)

TRANSITION? -- update node implementation info

MESSAGE add_specification(node)

TRANSITION? -- update specification info

MESSAGE add_control_constraints(node)

TRANSITION? -- update control constraints

END

The result is a lower level set of definitions for the system concepts and interfaces. These messages reveal operations on the abstract data type node. The lower level messages in the user_interface interface are:

add_graphic_record

add_implementation

add_specification

add_control_constraints

The addition of these messages creates an additional function requirement:

- Create and maintain version control.

A new node could be created when a nodes' properties are significantly changed by these low level messages. The ability to define a significant change will be required. The current implementation of the DDB does not address this function. Chapter IV does contain a discussion of version control.

As evidenced by the functions required of the DDB, a conventional DBMS will not suffice. Therefore, an object-oriented DBMS will be used to design and implement the DDB. The object-oriented DBMS that will be used is Vbase by Ontologic Incorporation.

IV. FEASIBILITY FOR THE DESIGN DATABASE USING VBASE

A. EXAMPLES OF COMPONENTS IN THE DESIGN DATABASE

1. Type Definition

As stated earlier, the first steps in a Vbase design are to identify the objects, their properties, and the frequent operations performed on the objects. The next step is to define the objects in TDL. The only object in the Design Database is a Node. The properties of a node were defined earlier as well. The frequent operations are those necessary to assist in the accomplishment of the required functions of the design database. The TDL code below illustrates the definition of a Node.

```
define Type Node
    supertypes = {Entity}
    properties = {
        name: String;
        specification: String;
        implementation: optional String;
        controlconstraints: optional String;
        graphicrecord: optional String;
        subNodes: distributed Set[Node] inverse $Node$isChildOf;
        isChildOf: optional Node inverse $Node$subNodes;
        }
```

The main points to notice about the above definition are the supertypes, optional, and inverse keywords. The supertypes declaration is used to show the parent class of a type. This is used for inheritance purposes. The supertype Entity is the root of all types in the Vbase system type library [Ref. 17:p. IX-6]. Entity specifies basic behavior for all object types in the system.

The keyword optional specifies that a property need not necessarily have a value. This indicates that a PSDL specification may or *may not have an implementation*, control constraints, or graphic record. The implementation property is optional due to the process through which *PSDL* specifications are created. The *control* constraints and graphic record are true optional properties in that they may or may not ever exist depending on whether implementation is graphic or ADA.

The inverse property sets up a system-maintained relationship between the property defined and its specified inverse. This property is used to maintain the parent-child relationship between Nodes. The inverse property also illustrates the "$" notation. The "$" notation is used to provide a mechanism for referring to names relative to their scope. The symbol "$" acts as a pathname separator.

The next step is to define the frequent operations on the object. The clause "operations = {...};" defines the set of operations that type Node will implement. The operations for a Node are buildDisplay, listsubNodes, listsubNodesInternal, and a refinement of the delete operation. BuildDisplay is used for output of a Node contents.

ListsubNodes and listsubNodesInternal are used to retrieve the children of a particular Node. The refines delete operation means the current definition is refining an operation already defined in the supertype. The actual refinement is achieved in the COP method which implements the operation and will be described later. The operations for a Node are defined as:

```
    operations = {

    buildDisplay (n:Node,)
      returns(Node)
      method (NodeBuildDisplay);

    listSubnodes (n:Node)
      returns (Set[Node])
      method (NodeListSubNodes);

    listSubnodesInternal (n:Node, s:Set [Node])
      returns (Set[Node]
      method (NodeListSubNodesInt);

    refines delete (n:Node)
      raises (CannotDelete)
      triggers (NodeDeleteTrigger);

};
```

The definition of a Node includes two procedure definitions:

define Procedure Create ... end Create;

and

define Procedure Lookup ... end Lookup;

Procedures differ from typed operations in that they are not tied to a type. For example, the operation $Node$buildDisplay can only be called on instances of type Node, while the procedure $Node$Lookup can be called with any arguments which satisfy the argument type specification of the procedure. The procedure Create has an argument specification of the form:

```
define Procedure Create
(t:Type,
    keywords
    name: String,
      specification: String,
      optional implementation: String,
      optional controlconstraints: String,
      optional graphicrecord: String,
      optional isChildOf: Node,
      optional where: Entity,
      optional hownear: Clustering
)
returns (Node)
raises (NodeAlreadyExists)
triggers (NodeCreateTrigger)
supertypes = {$Entity$Create};
end Create;
```

This specification gives more examples of the power of Vbase. Specifically, the keywords "keywords", "raises", "triggers", and "hownear". The keyword "keywords" specifies that the remaining arguments are passed by keywords, rather than by position in the argument list.

The statement "raises (NodeAlreadyExists)" specifies that the procedure may raise the exception NodeAlreadyExists. This exception is to alert the caller that the Node already exists, rather than creating a new copy of the Node.

The Create procedure also has a triggers clause, "triggers = (NodeCreateTrigger)". The Vbase system automatically generates a Create Procedure for every type defined. An explicit definition is required to specify a trigger to the system-defined Create. A trigger is a method associated with the invocation of a procedure or operation. Whenever the Create procedure is invoked, the trigger, NodeCreateTrigger, will be executed first. The trigger checks whether a Node already exists before creating a new one. An operation can have more than one trigger associated with it. The triggers are invoked in the order they are listed, and the last trigger must invoke the base method. The base method is the method which is specified as implementing the operation or procedure. [Ref. 17:p. 10-11]

The optional keywords where and hownear can be used in the Create operation to optimize disk storage of the object created to improve database performance. The type Clustering is an enumerated type with three instances: $area, $segment, and $chunk. Each is a unit of storage on the disk. Segment is the atomic unit of transfer from disk to the main memory cache. To specify that an object created is to be stored in the same segment as some other object, the value of the hownear argument is "$segment" and the value of the where argument is the other object. $Area and $chunk are not currently supported. [Ref. 17:p. IX-41]

The second procedure defined is called Lookup. This procedure is used to look up a given Node, identified by its Node name, in NodeCatalog which is a global Node catalog. The specification for procedure Lookup is as follows:

```
define Procedure Lookup (s:String)
returns (Node)
raises (NodeNotInCatalog)
method (NodeLookup)
supertypes = {Entity};
end Lookup;
```

There is one additional TDL definition in the DDB, NodesExceptions.tdl. NodesExceptions contains the definitions of the exceptions used in the application. The complete TDL definitions are listed in Appendix C.

2. COP Definition

The next step in the design of a Vbase application is to implement the frequent operations using COP. COP is an object-based superset of the language C. This can be either an advantage or a disadvantage of Vbase, depending on the designers knowledge of the C language. One method implemented for the object Node was "NodebuildDisplay". This operation is used to output the contents of a Node to a file. The COP code below implements the method:

```
#include <stdio.h>   /* include standard C routines */
#include <string.h>
#define MAXLINE 81   /* maximum line length is 81 characters */
#define MAXSTRING 4000 /* maximum string length is 4000 */
char opname[MAXLINE];    /* declaration of local variables */
FILE *out;
char spectext[MAXSTRING],
     imptext[MAXSTRING],
     cctext[MAXSTRING];

import $Type;
import $Class;
enter module $Node;

method
obj $Node
NodeBuildDisplay(aNode)
obj $Node aNode;
{
   out = fopen("ddb.out", "a");
    /* convert object code to C code */

   AM_stringToC(aNode.name,opname,sizeof(opname));
   fprintf(out,"%s\n",opname);
   AM_stringToC(aNode.specification, spectext, sizeof(spectext));
   fprintf (out,"%s\n", spectext);

   /* determine if optional property has a value */
   /*    before executing an operation on it.    */

   if (hasvalue(aNode.implementation))
   {
   AM_stringToC(aNode.implementation, imptext, sizeof(imptext));
   fprintf(out, "%s\n", imptext);
   }
   if (hasvalue(aNode.controlconstraints))
   {
   AM_stringToC(aNode.controlconstraints, cctext, sizeof(cctext));
   fprintf(out, "%s\n", cctext);
   }
    fclose(out);
   return(aNode);
 }
```

This example helps to show the ability to interweave the standard C language with COP. Object code and variables are prefaced by the "$" symbol. This is used to distinguish object variables from standard C variables.

The declarative statements "import" and "enter module" are used for name visibility. Making a name visible provides the COP compiler with a reference to what the name defines. Database names are defined in the Vbase Kernel Database and in TDL code. Names defined in the Vbase Kernel Database are globally defined in a default set. Names not included in the default set must be made visible explicitly using the "import" and "enter module" statements. An "import" declaration imports the definitions of a set of database names so they are visible within the current COP compilation unit. An "enter module" declaration establishes visibility for all names defined in a module. [Ref. 17:p. VII-3]

The functions "hasvalue" and "AM_stringToC" are system supplied. The function "hasvalue" is used to test whether an optional property has a value before executing any operations on it. This is necessary because of the strong-typing of Vbase. The function "AM_stringToC" is used to convert from an object string to a standard C string for use by systems external to the Vbase database.

Two other operations defined where "NodeListSubNodes" and "NodeListSubNodesInternal". These operations are used to assist in the traversal of the tree structure. The COP code to implement these operations is shown below:

```
method
obj $Set[obj $Node]
NodeListSubNodes(aNode)
obj $Node aNode;
{
   obj $Set[obj $Node] theSubNodes;
   theSubNodes = $Set[obj $Node]$[];
   $Node$ListSubNodesInternal(aNode, theSubNodes);
   return(theSubNodes);
}

method
obj $Set[obj $Node]
NodeListSubNodesInt(aNode, theSubNodes)
obj $Node aNode;
obj $Set[obj $Node] theSubNodes;
{
   obj $Node currentNode;
   iterate(currentNode = aNode.SubNodes)
   {
   $Set$Insert(theSubNodes, currentNode);
   $Node$ListSubNodesInternal(currentNode, theSubNodes);
   }
   return(theSubNodes);
}
```

This code helps to demonstrate other powerful capabilities of Vbase. One is the ability of one method to invoke another method. This is shown in the method "NodeListSubNodes". The other capability is the system supplied iterator operation. This operation is used to control aggregate types. The system defined iterator can be modified to return the aggregate in any order the user decides.

The remaining operations and methods are listed in Appendix C. The above was shown to demonstrate the feasibility and power of Vbase.

3. Application Programs

The final step in a Vbase design is to develop C or COP user applications. The user applications developed correspond to the functional specifications and architectural design. The user applications developed in response to the functional specifications are:

- CreateRootNode. Used to create a Node which is the root of a tree.
- CreateChildNode. Used to create a Node which is a child of a Node.
- GetParent. Used to retrieve the name of a Node's parent Node.
- GetChildren. Used to retrieve the name(s) of a Node's child Node(s).
- DeleteNode. Used to delete a Node and it's children from the tree.
- TraverseTree. Used to traverse the entire tree structure to generate the PSDL program.

The user applications developed in response to the architectural design are:

- StoreProperty. Used to update, insert, or change the value of a Node's property.
- GetProperty. Used to retrieve the contents of a Node's property.

The applications were all implemented using COP. They are shown in Appendix D. The actual code for some of these applications is quite long, therefore the code for TraverseTree will be shown here for demonstration purposes. This application takes as input the name of the root Node of a tree. It then iterates through the entire tree writing the properties of a Node to the output file "ddb.out".

```
#include <stdio.h>  /* include standard C routines */
#include <string.h>
#define LINELENGTH 80
#define MAXLINE 81 /* maximum linelength is 81 characters */
FILE *in, *out;  /* local variable declarations */
char rootname[MAXLINE],
    tempname[MAXLINE];
import $Node;
main(argc, argv)
int argc;
char **argv;
{
   /* local object variables */
   char *dbname;
   char *getenv();
   obj $Node theNode, currentNode;
   obj $Set[obj $Node] theNodes;
   obj $String theRoot;
   if (argc > 1)
      {
        dbname = argv[1];
      }
   else if (dbname = getenv("DBNAME"))
      {
      }
   else
      {
        printf("Must specify database name, either as a command
        line argument,\nor via the Unix environment variable
        DBNAME\n");
        exit(1);
      }
    {
      AM_databaseOpen(dbname, 0);
      in = fopen("ddb.in", "r");
      out = fopen("ddb.out", "w");
      fclose(out);
      fgets(tempname, LINELENGTH, in);
      strncpy(rootname, tempname, (strlen(tempname) - 1));
      theRoot = rootname;
      theNode = $Node$Lookup(theRoot);

      (void) $Node$BuildDisplay(theNode);

      theNodes = $Node$listSubnodes(theNode);
      iterate(currentNode = theNodes)
      {
       (void) $Node$BuildDisplay(currentNode);
      }
    }
      protect
        AM_databaseClose(dbname);
}
```

The above code illustrates two additional keywords: "void" and "protect". "Void" is a standard C keyword, indicating that the method does not return a value. The method simply outputs the information passed to it.

"Protect" ensures execution of a statement when an exception is raised. "Protect AM_databaseClose(dbname)" ensures the database involved is closed in the event an exception is raised.

4. Interface Requirements of the DDB and User Interface

The current implementation of the DDB is invoked automatically by the User Interface. The design engineer is not required to have any knowledge of the DDB. For example, whenever a designer creates a new root Node, the User Interface will automatically retain the needed information and invoke the "createRootNode" application. The same is true for all the other operations of the DDB. An excellent followup thesis to this one can concentrate on implementing a graphical interface between the two systems to allow manual and automatic operation invocation. The goal of the system should be to make the underlying system transparent to the user.

Input and output between the DDB and User Interface is accomplished through the use of two files titled "ddb.in" and "ddb.out". The input format required for each application is described below and is maintained by the User Interface.

(1) Create Root Node - used to create a root node.

Format required for ddb.in

- node name
- specification
- implementation (optional)
- control constraints (optional)

(2) Create Child Node - used to create a child node.

Format required for ddb.in

- node name
- parent node name
- specification
- implementation (optional)
- control constraints (optional)

(3) Store Property - used to store or change a node property.

Format required for ddb.in

- node name
- property

(4) Get Property - used to retrieve a node property.

Format required for ddb.in

- node name
- property name

output to ddb.out

- property

(5) Get Children - used to retrieve the names of a parent nodes children.

Format required for ddb.in

- parent node name

output to ddb.out

- child node(s) names

(6) Get Parent - used to retrieve the name of a child nodes parent.

Format required for ddb.in

- child node name

output to standard output device

- parent node name

(7) Delete Node - used to delete a node and all of it's children.

Format required for ddb.in

- node name to delete

(8) Traverse Tree - used to traverse the entire tree and produce the psdl program.

Format required for ddb.In

- root node name

output to ddb.Out

- entire psdl program

5. Version Control

The version control function has not been addressed to date. The previous functions were implemented without concern for version control. However, the issue of version control is an important one and must be addressed in future implementations.

A pre-release article from Ontologic, Inc. describes a history mechanism embedded into the Vbase Object Manager [Ref. 20:p. 1]. However, this feature has not been implemented in current releases. A discussion of the key points raised in the article will be helpful to further explain the issue of version control. Hopefully, future Vbase releases will incorporate this feature.

There are two basic approaches to version control: linear and non-linear evolution. Linear evolution is a series of states through which an entity passes as it is mutated [Ref. 20:p. 1]. A version is a snapshot of the entity at a point in time. The whole set of versions, called its Version-Set, represents the entire history of an entity [Ref. 20:p. 2]. Figure 6 gives a graphical representation of the Version-Set.

V: {v1 -->v2 -->v3 -->v4}

Figure 6. Version Set

Non-linear evolution is defined as a situation in which a version has more than one successor or more than one predecessor version [Ref. 20:p. 5]. In order to correctly maintain these relationships, there must be a method to describe the default path from a version. The other required method is to describe alternatives to the default path. The system must resolve a simple reference by default to the most recent version in a linear evolution path, and to the default branch of a forking evolution path, so simple references are unambiguous [Ref. 20:p. 5].

There are two ways to make a new version: manually or automatically. Manual version creation can be invoked by the user whenever he thinks that a significant change has occurred. Automatic version creation is invoked without the intervention of the user [Ref. 20:p. 3]. This is controlled by the type of the entity via its property and operations definitions.

In addition to the PSDL components of a Node, there must be audit trail information stored in the Node for version control. The current implementation of the DDB does not include audit trail information. The recommended audit trail information to be used in future implementations is given in Figure 7.

As previously stated, a system supplied version mechanism is not implemented in current release versions. The set of tools required to implement version control are within Vbase and could be implemented by the designer. This would be another excellent followup thesis to this one.

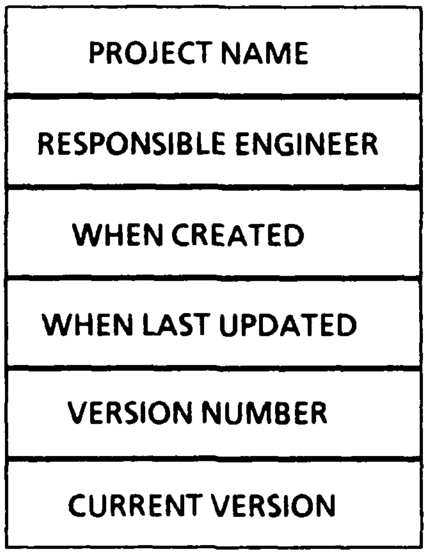

Figure 7. Audit Trail Properties

V. CONCLUSIONS AND RECOMMENDATIONS

A. SUMMARY AND CONCLUSIONS

The development of hard, real-time software systems continues to be an expensive process for the DOD. The Computer-Aided Prototyping System (CAPS) is one tool under development which will help to decrease the costs of these systems. CAPS is an attempt to integrate several of the prevailing software development methodologies into one tool. With a central theme of rapid prototyping, CAPS shows great promise for the future of software development in the DOD.

This thesis concentrated on the development of the Design Database (DDB) for CAPS. It is a key element of the system as project management has become an issue of increasing importance in software development. A robust Design Database which can efficiently and effectively, store and retrieve the Prototype System Description Language (PSDL) program will significantly contribute to the overall success of CAPS.

The goal of this thesis has been to develop a conceptual level design and initial implementation of the Design Database for CAPS. The basic design was developed using the object-oriented approach and the initial implementation was accomplished with an object-oriented DBMS (Vbase). Object-oriented technology offers several enhancements to current DBMS technology, and with its maturity it will become as important to CAD applications as relational database technology has become to business applications. This study has accomplished the goals of the thesis and identified key aspects of the design Database for continued implementation and follow-up thesis work.

B. RECOMMENDATIONS FOR FURTHER RESEARCH

This thesis has provided a foundation for further implementation of the Design Database. Further research and testing is required to complete full implementation of the system and to identify potential weaknesses. This author recommends work in the following specific areas:

- The design and implementation of version control within the design database. The issue of version control is an important one and must become an integral part of the DDB if it is to become an effective part of CAPS.

- The design and implementation of a graphical interface between the User Interface and the remainder of CAPS. Without an effective User Interface, CAPS will find little usefulness in the "real world".

- The study of memory management for the DDB as well as the entire system. Each application in Vbase requires two megabytes of memory. With the DDB and the SBMS both implemented in Vbase, the memory requirements for these two systems alone will be immense.

- The design and implementation of an efficient method for constraint checking within the various levels of decomposition. Currently, the User Interface is maintaining minimal information for this purpose but a more elaborate system is required.

APPENDIX A

REUSABLE COMPONENTS LIBRARY

The model was formulated in terms of reusable components for business applications. The reusable component library used is shown below as taken from Ref. 19. Explanatory comments extend from the leftmost "--" to the end of the line.

```
relation is_a(object,type)
  -- Means the object is an instance of the type.
 is_a(every object,a type)
  -- Every object is an instance of some type.

relation a_kind_of(type,type)
 a_kind_of(any type1,any type2) <=>
 (is_a(any object,type1) => is_a(object,type2))
  -- a_kind_of(type1,type2) means type1 is a subset of type2.

type object
 a_kind_of(any type,object)
  -- Any type is a subset of the universal type "object".

type type
  -- The set of all types.
 is_a(an object1,any object2) <=> is_a(object2,type)
  -- Any object with instances is a type and all types are non-empty.

type relation
  -- The set of all relations.
 ~(is_a(any object,type) & is_a(object,relation))
  -- Types and relations do not overlap.

relation unique(type)
  -- Means there is only one instance of the type.
 unique(any type) <=>
 is_a(any object1,type) & is_a(any object2,type) => equal(object1,object2)

type software_system
  -- The set of systems to be realized as programs.

relation proposed(software_system)
  -- Means the software system is going to be developed.
```

```
relation controls(software_system,type)
  -- Means the instances of the type are created, modified, and
  -- destroyed only by means of the software system.

type agent
  -- The cause of an activity, usually a person or organization.

type activity
  -- The set of all processes and actions.

type user
 a_kind_of(user,agent)
  -- A class of users of a software_system.
  uses(every user,a software_system)

relation uses(user,software_system)
  -- Means the user depends on the software_system to achieve some
  -- goals.

relation maintains(user,type)
  -- Means the instances of the type are created, modified, or
  -- destroyed upon request from the user.
 maintains(any user,any type) & controls(any software_system,type) =>
  uses(user,software_system)

type supplier
 a_kind_of(supplier,agent)
 supplies(every supplier,an object)

type vendor
 a_kind_of(vendor,supplier)
 sells(every vendor,a product)

type customer
 a_kind_of(customer,agent)
 buys(every customer,a product)

relation buys(customer,object)
  -- Means the customer buys instances of the object.
 buys(any customer,any object) => buys_from(customer,object,a vendor)

relation buys_from(customer,object,vendor)
  -- Means the customer buys the object from the vendor.
 buys_from(any customer,any object,any vendor) =>
pays(customer,vendor)
```

```
relation pays(customer,vendor)
  -- Means the customer sends money to the vendor.

relation supplies(supplier,object)
  -- Means the supplier creates and delivers instances of the object.

relation sells(vendor,object)
  -- Means the vendor sells instances of the object.
 sells(any vendor,any object) => supplies(vendor,object)
  -- To sell something, a vendor must supply it.
 sells(any vendor,any object) => buys(a customer,object)
  -- To sell something, someone must buy it.

relation needed_for(object,activity)
  -- Means an instance of the object is needed for the activity to occur.

relation wants(agent,object)
  -- Means the agent is motivated to acquire the object.
 wants(any agent,any activity) & needed_for(any object,activity) =>
  wants(agent,object)
  -- People want the means to achieve their ends.
 wants(any customer,any object) => buys(customer,object)
  -- Customers are agents with the resources to satisfy their wishes.
```

APPENDIX B

MODEL LANGUAGE GRAMMAR

The grammar for the Model language used in the requirements analysis is shown below as taken from Ref. 19. Terminal symbols appear in double quotes. Repetitions are indicated by x* (zero or more x's) or x+ (one or more x's). Alternatives are separated by vertical bars. Ranges of single character alternatives are shown [0-2] (meaning "0" | "1" | "2").

```
model = (type | relation)*

type = "type" name law* attribute*
relation = "relation" name "("name_list")" law*
attribute = name "("name_list")" ":" name
law = relationship | "-" law | law op law | "(" law ")"
relationship = name "(" arg_list")"
op = "&" | "|" | "=>" | "<=>"

name_list = name ("," name)*
arg_list = arg ("," arg)*
arg = variable | attribute_value
variable = prefix name digit* dependency
attribute_value = arg "." name dependency
prefix = "a" | "an" | "every" | "any" | ""
dependency = "("arg_list")" | ""

name = alpha+ ("_" alpha+)*
alpha = [a-z] | [A-Z]
digit = [0-9]
```

All ops are left associative: a op b op c means (a op b) op c.
Precedence order: (strongest) ~, &, |, =>, <=> (weakest).
Comments extend from the leftmost "--" to the end of the line.

APPENDIX C

TDL DEFINITION EXAMPLES

The TDL definitions have all been successfully compiled and tested. Testing consisted of actual operation of the Design Database using an example prototype. The tests were conducted by invoking each application or operation with the correct format in the input file "ddb.in". All exception definitions were tested by intentionally invoking each operation.

Node definition

```
define Type Node

  supertypes = {Entity};

  properties =
  {
  name: String;
  specification: String;
  implementation: optional String;
  controlconstraints: optional String;
  graphicrecord: optional String;
  subNodes: distributed Set[Node] inverse $Node$isChildOf;
  isChildOf: optional Node inverse $Node$subNodes;
  };

  operations =
  {
  buildDisplay (n:Node)
   returns(Node)
   method (NodeBuildDisplay);

  listSubnodes (n:Node)
   returns (Set[Node])
   method (NodeListSubNodes);

  listSubnodesInternal (n:Node,s:Set[Node])
   returns (Set[Node])
   method (NodeListSubNodesInt);
```

```
 refines delete (n: Node)
 raises (CannotDelete)
 triggers(NodeDeleteTrigger);

};

define Procedure Create

 (t:Type,

 keywords
   name: String,
   specification: String,
   optional implementation: String,
   optional controlconstraints: String,
   optional graphicrecord: String,
   optional isChildOf: Node,
   optional where: Entity,
   optional hownear: Clustering)

 returns (Node)
 raises (NodeAlreadyExists)

 triggers (NodeCreateTrigger)

 supertypes = {$Entity$Create};
end Create;

define Procedure Lookup (s:String)
 returns (Node)
 raises (NodeNotInCatalog)
 method (NodeLookup)
 supertypes = {Entity};
end Lookup;

end Node;

define UnorderedDictionary NodeCatalog
 memberspec = Node;
 indexSpec = String;
end NodeCatalog;

define Variable NodeCatalogVar: UnorderedDictionary[Node, String] :=
NodeCatalog;

define variable NodeSerialNumber: Integer := 1;
```

Exception definitions

```
define ExceptionType NodeException

  supertypes = {Exception};
  properties =
  {
  nodeName: String;
  };
end NodeException;

define ExceptionType NodeNotInCatalog

  supertypes = {NodeException};
end NodeNotInCatalog;

define ExceptionType NodeAlreadyExists

  supertypes = {NodeException};
end NodeAlreadyExists;
```

APPENDIX D

COP OPERATIONS CODE

The COP definitions have all been successfully compiled and tested. Testing consisted of actual operation of the Design Database using an example prototype. The tests were conducted by invoking each operation with the correct format in the input file "ddb.in". All exception definitions were tested by intentionally invoking each operation.

String Definitions

```
/* This file contains all the symbolic constant definitions used */
/* in the COP operation and application programs */

#define LINELENGTH 80 /* Linelength is 80 characters */
#define MAXLINE 81 /* Maximum linelength is 81 characters */
                  /* including the null (\0) character */
#define LINEBUFFER 82 /* Temporary buffer used to hold Maxline */
#define MAXSTRING 4000 /* Maximum length of a String in Vbase */
```

Node Methods

```
/* Program - Node.c */
/* This program implements the methods defined in Node.tdl */
/* The methods implemented are */
/* Node Build Display          */
/* Node List SubNodes          */
/* Node List SubNodes Internal */
/* Node Lookup                 */
/* Node Create Trigger         */
/* Node Delete Trigger         */

#include <stdio.h>  /* include standard C routines */
#include <string.h>
#include "string.def" /* include string definitions file */
char opname[MAXLINE]; /* Local variable declarations */
FILE *out;
char spectext[MAXSTRING],
     imptext[MAXSTRING],
     cctext[MAXSTRING];
```

```
import $Type;
import $Class;

enter module $Node;

method
obj $Node
NodeBuildDisplay(aNode)

obj $Node aNode;
{
   out = fopen("ddb.out", "a");

   /* Convert Object Code to C Code */
   AM_stringToC(aNode.name,opname,sizeof(opname));
   fprintf(out,"%s\n",opname);
   AM_stringToC(aNode.specification,spectext,sizeof(spectext));
   fprintf (out,"%s\n", spectext);


   if (hasvalue(aNode.implementation))
   {
   AM_stringToC(aNode.implementation, imptext, sizeof(imptext));
   fprintf(out, "%s\n", imptext);
   }
   if (hasvalue(aNode.controlconstraints))
   {
   AM_stringToC(aNode.controlconstraints, cctext,  sizeof(cctext));
   fprintf(out, "%s\n", cctext);
   }
   fclose(out);
   return(aNode);
}

method
obj $Set[obj $Node]
NodeListSubNodes(aNode)
obj $Node aNode;
{
   obj $Set[obj $Node] theSubNodes;
   theSubNodes = $Set[obj $Node]$[];
   $Node$ListSubNodesInternal(aNode, theSubNodes);
   return(theSubNodes);
}
```

```
method
obj $Set[obj $Node]
NodeListSubNodesInt(aNode, theSubNodes)
obj $Node aNode;
obj $Set[obj $Node] theSubNodes;
{
   obj $Node currentNode;
   iterate(currentNode = aNode.SubNodes)
{
   $Set$Insert(theSubNodes, currentNode);
   $Node$ListSubNodesInternal(currentNode, theSubNodes);
}
   return(theSubNodes);
}

method
obj $Node
NodeLookup(aNodeName)
obj $String aNodeName;
{
   obj $Node theNode;
   theNode =
   $UnorderedDictionary$GetElement($NodeCatalogVar,aNodeName);
   except(enf:ElementNotFound)
{
   raise NodeNotInCatalog(nodeName: aNodeName);
}
   return(theNode);
}
method
obj $Node
NodeCreateTrigger(aType, name, specification, implementation,
                  controlconstraints, graphicrecord,
                  isChildOf, where, hownear)

obj $Type aType;
keyword obj $String name;
keyword obj $String specification;
keyword obj $String implementation;
keyword obj $String controlconstraints;
keyword obj $String graphicrecord;
keyword obj $Node isChildOf;
keyword obj $Entity where;
keyword obj $Clustering hownear;
```

```
{
obj $Node theNode;
{
   theNode = $Node$Lookup(name);
   raise NodeAlreadyExists(nodename:name);

}
   except (nnc: NodeNotInCatalog)
{
   theNode = $$(aType, name:name, specification:specification,
                implementation:implementation,
                controlconstraints:controlconstraints,
                graphicrecord:graphicrecord,
                isChildOf:isChildOf,
                where:where, hownear:hownear);

   $UnorderedDictionary$Insert($NodeCatalogVar,name,theNode);
   return(theNode);
}
}

method
void
NodeDeleteTrigger(aNode)

obj $Node aNode;
{
obj $Node theNode;
iterate(theNode = aNode.subNodes)
$UnorderedDictionary$Remove($NodeCatalogVar,theNode.name);

$$(aNode);
}
```

APPENDIX E

APPLICATION PROGRAMS

The application programs have all been successfully compiled and tested. Testing consisted of actual operation of the Design Database using an example prototype. The tests were conducted by invoking each application or operation with the correct format in the input file "ddb.in". All exception definitions were tested by intentionally invoking each operation.

Create Root Node

```
/* Program CreateRootNode */
/* This program is used to create a Node which will be the root of */
/* a tree structure. It interfaces with the User Interface through */
/* a file called ddb.in.                                         */
/* The program reads in the required information from ddb.in        */
/* then creates the Node and inserts it as the root node          */
/* The required information in ddb.in is */
/* Node Name                          */
/* Specification                      */
/* Implementation (Optional)          */
/* Control Constraints (Optional)     */

#include <stdio.h>   /* include Standard C routines */
#include <string.h>
#include "string.def" /* include string definitions file */
/* Local variable declarations */
char tempname[MAXLINE],
    opname[MAXLINE],
    templine[MAXLINE],
    tempgr[MAXLINE];

char Temptext[LINEBUFFER],
    Graphtext[LINEBUFFER];

char Spectext[MAXSTRING],
    Imptext[MAXSTRING],
    Cctext[MAXSTRING],
    Grlink[MAXSTRING];
```

```
FILE *in, *gr;
int linelength;
int i;
import $Node;


main (argc, argv)
int argc;
char **argv;
{
   char *dbname;
   char *getenv();
   obj $Node theNode;  /* local object variables */
   obj $String thename,
            thespec,
            theimp,
            thecc,
            thegraph;

if (argc > 1)
   {
     dbname = argv[1];
   }
else if (dbname = getenv("DBNAME"))
   {
   }
else
   {
  printf("Must specify database name, either as a command line
  argument,\nor via the Unix environment variable DBNAME\n");
  exit(1);
   }
{
AM_databaseOpen (dbname, 0);
in = fopen("ddb.in", "r");
fgets(tempname, LINELENGTH, in); /* Read the name of the Node */
strncpy(opname, tempname, (strlen(tempname) - 1));
fgets(templine, LINELENGTH, in); /* Read first property */
if (strncmp("SPECIFICATION", templine, 13) == 0)
/* Verify Specification is first Property */
{
   strncpy(Spectext,templine, strlen(templine));
   strcat(Spectext, "\n");
   templine[0] = '\0';
   for(i = 0; i < MAXLINE; i++)
      Temptext[i] = '\0';
   fgets(templine, LINELENGTH, in);
   while(strncmp("IMPLEMENTATION",templine, 14) != 0)
/* Read in Specification Until Implementation Begins */
   {
      strncpy(Temptext, templine,strlen(templine));
      strcat(Temptext, "\n");
      strcat(Spectext, Temptext);
      templine[0] = '\0';
      for(i = 0; i < MAXLINE; i++)
         Temptext[i] = '\0';
      fgets(templine, LINELENGTH, in);
   }
   Cctext[0] = '\0';
   Grlink[0] = '\0';
   strncpy(Temptext, templine, strlen(templine));
   strcat(Temptext, "\n");
   strcat(Imptext, Temptext);
   templine[0] = '\0';

   for(i = 0; i < MAXLINE; i++)
      Temptext[i] = '\0';
   fgets(templine, LINELENGTH, in);

/* If Implementation is graph then read in graphic record */

   if (strncmp("GRAPH", templine, 5) == 0)
   {
      gr = fopen("links.c", "r");
      fgets(tempgr, LINELENGTH, gr);
      while (feof(gr) == 0)
      {
         strncpy(Graphtext, tempgr, strlen(tempgr));
         strcat(Graphtext, "\n");
         strcat(Grlink,Graphtext);
         tempgr[0] = '\0';
         for (i = 0; i < MAXLINE; i++)
            Graphtext[i] = '\0';
         fgets(tempgr, LINELENGTH, gr);
      }
    }
    fclose(gr);
    strncpy(Temptext, templine, strlen(templine));
    strcat(Temptext, "\n");
    strcat(Imptext, Temptext);
    templine[0] = '\0';
    for (i = 0; i < MAXLINE; i++)
       Temptext[i] = '\0';
      while (feof(in) == 0)
      {
         fgets(templine, LINELENGTH, in);
         if(strncmp("CONTROL CONSTRAINTS", templine, 19) == 0)
      /* Read in Control Constraints */
         {
         while(feof(in) == 0)
         {
            strncpy(Temptext, templine, strlen(templine));
            strcat(Temptext, "\n");
            strcat(Cctext, Temptext);
            templine[0] = '\0';
            for (i =0; i < MAXLINE; i++)
               Temptext[i] = '\0';
            fgets(templine, LINELENGTH, in);
          }
          }
          else

     /* Read in Implementation */
         {
             strncpy(Temptext, templine, strlen(templine));
             strcat(Temptext, "\n");
             strcat(Imptext, Temptext);
             templine[0] = '\0';
             for (i = 0; i < MAXLINE; i++)
                Temptext[i] = '\0';
             fgets(templine, LINELENGTH, in);
           }
      }
   }
   else
   {
      printf("Input file is not in the correct format.");
      exit(1);
   }
   fclose(in);
   /* Assign C variables to Object Variables */
   thename = opname;
   thespec = Spectext;
   theimp = Imptext;
   /* Execute Create depending on the properties */
   if ((strlen(Cctext) == 0) && (strlen(Grlink) == 0))
   {
      theNode = $Node$[name:thename,specification:thespec,
                        implementation:theimp];
      except(nae:NodeAlreadyExists)
   {
      theNode = $Node$Lookup(thename);
   }
}
else if ((strlen(Cctext) == 0) && (strlen(Grlink) != 0))
{
   thegraph = Grlink;
   theNode = $Node$[name:thename, specification:thespec,
             implementation:theimp, graphicrecord:thegraph];
   except(nae:NodeAlreadyExists)
   {
      theNode = $Node$Lookup(thename);
   }
}
else
{
   thecc = Cctext;
   theNode = $Node$[name:thename, specification:thespec,
             implementation:theimp, controlconstraints:thecc];
   except(nae:NodeAlreadyExists)
   {
      theNode = $Node$Lookup(thename);
   }
}
}
   protect
      AM_databaseClose(dbname);
}
```

Create Child Node

```
/* Program Create Child Node */
/* This program is used to create a Node which will a child of */
/* a node in the tree structure. It interfaces with the User  */
/* Interface through a file called ddb.in.                 */
/* The program reads in the required information from ddb.in  */
/* then creates the Node and inserts in the proper order      */
/* The required information in ddb.in is */
/* Node Name                         */
/* Parent Node Name                  */
/* Specification                   */
/* Implementation (Optional)         */
/* Control Constraints (Optional)    */
```

```
#include <stdio.h>  /* include standard C routines */
#include <string.h>
#include "string.def" /* include string definitions file */
/* local variables */
char tempname[MAXLINE],
    opname[MAXLINE],
    templine[MAXLINE],
    tempgr[MAXLINE];
char tempparent[MAXLINE],
    parentname[MAXLINE],

char Temptext[LINEBUFFER],
    Graphtext[LINEBUFFER];

char Spectext[MAXSTRING],
    Imptext[MAXSTRING],
    Cctext[MAXSTRING],
    Grlink[MAXSTRING];

FILE *in, *gr;
int linelength;
int i;
import $Node;
main (argc, argv)

int argc;
char **argv;

{
   char *dbname;
   char *getenv();

   obj $Node theNode,
          parentNode; /* local object Variables */

   obj $String thename,
             thespec,
            theimp,
            thecc,
            thegraph,
            theparent;

if (argc > 1)
   {
     dbname = argv[1];
   }
else if (dbname = getenv("DBNAME"))
```

```
    {
    }
else
    {

 printf("Must specify database name, either as a command line
argument,\nor via the Unix environment variable DBNAME\n");
    exit(1);
    }
 {
AM_databaseOpen (dbname, 0);
in = fopen("ddb.in", "r");
fgets(tempname, LINELENGTH, in); /* Read the name of the Node */
strncpy(opname, tempname, (strlen(tempname) - 1));
fgets(tempparent, LINELENGTH, in);  /* Read the parent Node name */

strncpy(parentname, tempparent, (strlen(tempparent) - 1));
fgets(templine, LINELENGTH, in); /* Read in the property */
/* Verify Specification is first Property */
if (strncmp("SPECIFICATION", templine, 13) == 0)
    {
        strncpy(Spectext,templine, strlen(templine));
        strcat(Spectext, "\n");
        templine[0] = '\0';
        for(i = 0; i < MAXLINE; i++)
            Temptext[i] = '\0';
        fgets(templine, LINELENGTH, in);

        while(strncmp("IMPLEMENTATION",templine, 14) != 0)
        /* Read in Specification Until Implementation Begins */
    {
            strncpy(Temptext, templine,strlen(templine));
            strcat(Temptext, "\n");
            strcat(Spectext, Temptext);
            templine[0] = '\0';
            for(i = 0; i < MAXLINE; i++)
                Temptext[i] = '\0';
            fgets(templine, LINELENGTH, in);
        }
        Cctext[0] = '\0';
        Grlink[0] = '\0';
        strncpy(Temptext, templine, strlen(templine));
        strcat(Temptext, "\n");
        strcat(Imptext, Temptext);
        templine[0] = '\0';
        for(i = 0; i < MAXLINE; i++)
            Temptext[i] = '\0';
```

```
/* If Implementation is graph then read in graphic record */
    fgets(templine, LINELENGTH, in);
     if (strncmp("GRAPH", templine, 5) == 0)
        {
          gr = fopen("links.c", "r");
          fgets(tempgr, LINELENGTH, gr);
          while (feof(gr) == 0)
          {
             strncpy(Graphtext, tempgr, strlen(tempgr));
             strcat(Graphtext, "\n");
             strcat(Grlink,Graphtext);
             tempgr[0] = '\0';
             for (i = 0; i < MAXLINE; i++)
                Graphtext[i] = '\0';
             fgets(tempgr, LINELENGTH, gr);
          }
        }
     fclose (gr);
     strncpy(Temptext, templine, strlen(templine));
     strcat(Temptext, "\n");
     strcat(Imptext, Temptext);
     templine[0] = '\0';
     for (i = 0; i < MAXLINE; i++)
        Temptext[i] = '\0';

     while (feof(in) == 0)
     {
        fgets(templine, LINELENGTH, in);
        if(strncmp("CONTROL CONSTRAINTS", templine, 19) == 0)
       /* Read in Control Constraints */
     {
        while(feof(in) == 0)
        {
           strncpy(Temptext, templine, strlen(templine));
           strcat(Temptext, "\n");
           templine[0] = '\0';
           for (i =0; i < MAXLINE; i++)
              Temptext[i] = '\0';
           fgets(templine, LINELENGTH, in);
        }
        }
        else
        {
           strncpy(Temptext, templine, strlen(templine));
           strcat(Temptext, "\n");
           strcat(Imptext, Temptext);
           templine[0] = '\0';
```

```
            for (i = 0; i < MAXLINE; i++)
               Temptext[i] = '\0';
         }
      }
   }
   else
   {
      printf("Input file is not in the correct format.");
      exit(1);
   }
fclose(in);
/* Assign C variables to Object Variables */
theparent = parentname;
thename = opname;
thespec = Spectext;
theimp = Imptext;
parentNode = $Node$Lookup(theparent);
/* Execute Create depending on the properties */
if ((strlen(Cctext) == 0) && (strlen(Grlink) == 0))
{
   theNode = $Node$[name:thename,specification:thespec,
                    implementation:theimp, isChildOf:parentNode];
   except(nae:NodeAlreadyExists)


   {
      theNode = $Node$Lookup(thename);
   }
}
else if((strlen(Cctext) == 0) && (strlen(Grlink) != 0))
{
   thegraph = Grlink;
   theNode = $Node$[name:thename, specification:thespec,
            implementation:theimp, graphicrecord:thegraph,
            isChildOf:parentNode];
   except(nae:NodeAlreadyExists)
   {
      theNode = $Node$Lookup(thename);
   }
}
else
{
   thecc = Cctext;
   theNode = $Node$[name:thename, specification:thespec,
            implementation:theimp, controlconstraints:thecc,
            isChildOf:parentNode];
   except(nae:NodeAlreadyExists)
```

```
        {
           theNode = $Node$Lookup(thename);
        }
     }
  }
     protect
        AM_databaseClose(dbname);
  }
```

Store Property

```
/* Program Store Property */
/* This program is used to store or change a Node property.    */
/* It interfaces with the User Interface through a file called */
/* ddb.in.                                                      */
/* The program reads in the required information from ddb.in   */
/* then inserts the new property into the Node                  */
/* The required information in ddb.in is */
/* Node Name                            */
/* Property Name                        */

#include <stdio.h>  /* include standard C routines */
#include <string.h>
#include "string.def" /* include string definitions file */
/* local variables */
char opname[MAXLINE],
    tempname[MAXLINE],
    templine[MAXLINE];

char nodeproperty[MAXSTRING];
char temptext[LINEBUFFER];
FILE *in;
int i;
import $Node;

main(argc, argv)

int argc;
char **argv;

{
   char *dbname;
   char *getenv();
   obj $Node theNode; /* local object variables */
   obj $String theOperator,
            theProperty;
```

```
 if (argc > 1)
   {
      dbname = argv[1];
   }
else if (dbname = getenv("DBNAME"))
   {
   }
else
   {
      printf("Must specify database name, either as a command line
argument, \nor via the unix environment variable command");
      exit(1);
   }
 {
AM_databaseOpen (dbname, 0);
in = fopen("ddb.in", "r");
fgets(tempname, LINELENGTH, in); /* Read in the Node name */
strncpy(opname, tempname, (strlen(tempname) - 1));
fgets(templine, LINELENGTH, in); /* Read in the property name */
do
{
   strncpy(temptext, templine, (strlen(templine) - 1));
   strcat(temptext, "\n");
   strcat(nodeproperty, temptext);
   templine[0] = '\0';
   for(i = 0; i < MAXLINE; i++)
      temptext[i] = '\0';
   fgets(templine, LINELENGTH, in);
}
while(feof(in) == 0);
/* Assign c variables to object variables */
theOperator = opname;
theProperty = nodeproperty;
/* Verify the Node exists */
theNode = $Node$Lookup(theOperator);
/* Assign the property to its correct property */
if (strncmp("SPECIFICATION", nodeproperty, 13) == 0)
{
   theNode.specification = theProperty;

}
else if (strncmp("IMPLEMENTATION", nodeproperty, 14) == 0)
{
   theNode.implementation = theProperty;
}
else if (strncmp("CONTROL CONSTRAINTS", nodeproperty, 19) == 0)
{
   theNode.controlconstraints = theProperty;
}
else
{
   theNode.graphicrecord = theProperty;
}
fclose(in);
}
   protect
      AM_databaseClose(dbname);
}
```

Get Property

```
/*  Program Get Property */
/*  This program is used to retrieve a Node property.       */
/*  It interfaces with the User Interface through two files */
/*  ddb.in & ddb.out                                     */
/*  The program reads in the required information from ddb.in   */
/*  then outputs the requested information to ddb.out          */
/*  The required information in ddb.in is */
/*  Node Name                          */
/*  Property Name                      */
/*  The information output to ddb.out     */
/*  Property requested                 */


#include <stdio.h>  /* include standard C routines */
#include <string.h>
#include "string.def" /* include string definitions file */
/* local variables */
char opname[MAXLINE],
    tempname[MAXLINE],
    templine[MAXLINE];
char nodeproperty[MAXSTRING];
char temptext[LINEBUFFER];

FILE *in, *out;
int i;

import $Node;

main(argc, argv)

int argc;
```

```
char **argv;

{
   char *dbname;
   char *getenv();
   obj $Node theNode; /* local object variables */
   obj $String theOperator,
          theProperty;

if (argc > 1)
   {
      dbname = argv[1];
   }
else if (dbname = getenv("DBNAME"))
   {
   }
else
   {
      printf("Must specify database name, either as a command line
      argument, \nor via the unix environment variable command");
      exit(1);
   }
{
AM_databaseOpen (dbname, 0);
in = fopen("ddb.in", "r");
out = fopen("ddb.out", "w");
fgets(tempname, LINELENGTH, in); /* read in Node name */
strncpy(opname, tempname, (strlen(tempname) - 1));
fgets(templine, LINELENGTH, in); /* read in property name */
/* Assign C variable to object variable */
theOperator = opname;


 /* Verify Node exists */
theNode = $Node$Lookup(theOperator);
/* Determine correct property to output */
if (strncmp("SPECIFICATION", templine, 13) == 0)
{
   /* Assign property */
   theProperty = theNode.specification;
   /* Convert object type to C type */
   AM_stringToC(theProperty, nodeproperty, sizeof(nodeproperty));
   /* Output to ddb.out */
   fprintf(out, "%s\n", nodeproperty);
}
else if (strncmp("IMPLEMENTATION", templine, 14) == 0)
{
   /* Assign property */
   theProperty = theNode.implementation;
   /* Convert object type to C type */
   AM_stringToC(theProperty, nodeproperty, sizeof(nodeproperty));
   /* Output to ddb.out */
   fprintf(out, "%s\n", nodeproperty);
}
else if (strncmp("CONTROL CONSTRAINTS", templine, 19) == 0)
{
   /* Assign property */
   theProperty = theNode.controlconstraints;
   /* Convert object type to C type */
   AM_stringToC(theProperty, nodeproperty, sizeof(nodeproperty));
   /* Output to ddb.out */
   fprintf(out, "%s\n", nodeproperty);
}
else
{
   /* Assign property */
   theProperty = theNode.graphicrecord;
   /* Convert object type to C type */
   AM_stringToC(theProperty, nodeproperty, sizeof(nodeproperty));
   /* Output to ddb.out */
   fprintf(out, "%s\n", nodeproperty);
}
fclose(in);
fclose(out);
}
   protect
      AM_databaseClose(dbname);
}
```

Get Parent

```
/*  Program Get Parent */
/*  This program is used to retrieve the name of a Node's parent */
/*  It interfaces with the User Interface through ddb.in        */
/*  & the standard output device                                */
/*  The program reads in the required information from ddb.in    */
/*  then pipes the requested information to the standard output   */
/*  The required information in ddb.in is */
/*  Child Node Name                    */
/*  The information output to standard output */
/*  Parent Node Name                   */
```

```
#include <stdio.h> /* include standard c routines */
#include <string.h>
#include "string.def" /* include string definitions file */
/* local variables */
FILE *in;
char parentname[MAXLINE],
    tempname[MAXLINE],
    childname[MAXLINE];

import $Node;

main(argc, argv)

int argc;
char **argv;
{

   char *dbname;
   char *getenv();
   obj $Node theNode,
          parentNode; /* local object variables */
   obj $String theChild,
           theParent;

   if (argc > 1)
      {
     dbname = argv[1];
      }
   else if (dbname = getenv("DBNAME"))
      {
      }
   else
      {
printf("Must specify database name, either as a command line
argument,\nor via the Unix environment variable DBNAME\n");
         exit(1);
      }
   {
      AM_databaseOpen(dbname, 0);
      in = fopen("ddb.in", "r");
      fgets(tempname,LINELENGTH, in); /* read in Child Node Name */
      strncpy(childname, tempname, (strlen(tempname) - 1));
      /* Assign c variable to object variable */
      theChild = childname;
      /* Verify Node exists */
      theNode = $Node$Lookup(theChild);
      /* Assign Parent Name */
      parentNode = theNode.isChildOf;
      /* Convert to c code */
      AM_stringToC(parentNode.name, parentname, sizeof(parentname));
      /* Output to standard output device */
      printf("%s\n", parentname);
      fclose(in);
   }
      protect
         AM_databaseClose(dbname);
}
```

Get Children

```
/*  Program Get Children */
/*  This program is used to retrieve the name of a Node's     */
/*  Child or Children.  It interfaces with the User Interface */
/*  through ddb.in & ddb.out                                   */
/*  The program reads in the required information from ddb.in */
/*  then outputs the requested information to ddb.out         */
/*  The required information in ddb.in is */
/*  Parent Node Name                  */
/*  The information output to ddb.out     */
/*  Child Node Name(s)                */

#include <stdio.h>  /* include standard c routines */
#include <string.h>
#include "string.def" /* include string definitions file */

/* local variables */
FILE *in, *out;
char opname[MAXLINE],
    tempname[MAXLINE],
    childname[MAXLINE];

import $Node;

main(argc, argv)

int argc;
char **argv;
{
   char *dbname;
   char *getenv();  /* local object variables */
   obj $Node theNode,
           currentNode;
   obj $Set[obj $Node] theNodes;
   obj $String theParent;

   if (argc > 1)
      {
     dbname = argv[1];
      }
   else if (dbname = getenv("DBNAME"))
      {
      }
   else
      {
printf("Must specify database name, either as a command line
argument,\nor via the Unix environment variable DBNAME\n");
         exit(1);
      }

   {
      AM_databaseOpen(dbname, 0);
      in = fopen("ddb.in", "r");
      out = fopen("ddb.out", "w");
     fgets(tempname,LINELENGTH, in); /* read in parent node name */
      strncpy(opname, tempname, (strlen(tempname) - 1));
      /* assign to object variable */
      theParent = opname;
      /* verify node exists */
      theNode = $Node$Lookup(theParent);
      /* assign children names to variable */
      theNodes = theNode.subNodes;
      /* iterate through the set */
      iterate(currentNode = theNodes)
      {
      /* Convert object code to C code */
      AM_stringToC(currentNode.name, childname, sizeof(childname));
      /* Output to ddb.out */
      fprintf(out, "%s\n", childname);
      }
      fclose(in);
      fclose(out);
    }
      protect
         AM_databaseClose(dbname);
}
```

Delete Node

```
/* Program Delete Node */
/* This program is used to delete a Node from the tree      */
/* It also deletes any child Nodes coming from the Node     */
/* It interfaces with the User Interface through ddb.in     */
/* The program reads in the required information from ddb.in */
/* then deletes the node and all of its' children           */
/* The required information in ddb.in is */
/* Node Name                                  */

#include <stdio.h> /* include standard C routines */
#include <string.h>
#include "string.def" /* include string definitions file */
/* Local variables */
char opname[MAXLINE],
   tempname[MAXLINE];
FILE *in;
import $Node;

main(argc, argv)
int argc;
char **argv;

{
  char *dbname;
  char *getenv();
  obj $Node theNode; /* local object variables */
  obj $String theOperator;

if (argc > 1)
  {
     dbname = argv[1];
  }
else if (dbname = getenv("DBNAME"))
  {
  }
else
  {
     printf("Must specify database name, either as a command line
argument, \nor via the unix environment variable command");
     exit(1);
  }
{
AM_databaseOpen (dbname, 0);
in = fopen("ddb.in", "r");
fgets(tempname, LINELENGTH, in); /* read in the node name */
strncpy(opname, tempname, (strlen(tempname) - 1));
/* assign to object variable */
theOperator = opname;
/* verify node exists */
theNode = $Node$Lookup(theOperator);
/* delete the node by calling the delete operation */
$Node$Delete(theNode);
fclose(in);
}
  protect
    AM_databaseClose(dbname);
}
```

Traverse Tree

```
/*  Program Traverse Tree */
/*  This program is used to traverse the entire tree &      */
/*  produce the PSDL program.                              */
/*  It interfaces with the User Interface through ddb.in    */
/*  & ddb.out.                                           */
/*  The program reads in the required information in ddb.in */
/*  then traverses the entire tree outputting the contents  */
/*  of all nodes to ddb.out                                */
/*  The required information in ddb.in is */
/*  Root Node Name                    */
/*  The information outputted to ddb.out  */
/*  for each Node in the tree          */
/*  Node Name                          */
/*  Specification                     */
/*  Implementation (Optional)          */
/*  Control constraints (Optional)     */

#include <stdio.h> /* include standard C routines */
#include <string.h>
#include "string.def" /* include string definitions file */
/* local variables */
FILE *in, *out;
char rootname[MAXLINE],
   tempname[MAXLINE];

import $Node;

main(argc, argv)
int argc;
char **argv;
{
    char *dbname;
    char *getenv();
    /* local object variables */
    obj $Node theNode, currentNode;
    obj $Set[obj $Node] theNodes;
    obj $String theRoot;

    if (argc > 1)
       {
     dbname = argv[1];
       }
    else if (dbname = getenv("DBNAME"))
       {
       }
    else
       {
printf("Must specify database name, either as a command line
argument,\nor via the Unix environment variable DBNAME\n");
          exit(1);
       }

    {
       AM_databaseOpen(dbname, 0);
       in = fopen("ddb.in", "r");
       /* erase the contents of ddb.out */
       out = fopen("ddb.out", "w");
       fclose(out);
       fgets(tempname, LINELENGTH, in); /* read in root node name */
       strncpy(rootname, tempname, (strlen(tempname) - 1));
       /* assign to object variable */
       theRoot = rootname;
       /* verify Node exists */
       theNode = $Node$Lookup(theRoot);
       /* output root node contents */
       (void) $Node$BuildDisplay(theNode);
       /* Assign children of root to variable */
       theNodes = $Node$listSubnodes(theNode);
       /* iterate through all the children */
       iterate(currentNode = theNodes)
       {
        /* output the children node contents */
        (void) $Node$BuildDisplay(currentNode);
       }
    }
       protect
          AM_databaseClose(dbname);
}
```

Makefile

The makefile is not a part of the Design Database but is included to assist interested parties in the compilation and execution of the DDB. Due to current system memory requirements, the author does not anticipate leaving a compiled and working copy of the DDB at the Naval Postgraduate School. As stated in Chapter V, each application in Vbase requires two megabytes of memory. This fact requires that the author remove applications once compiled and tested. Therefore, the uncompiled programs will remain on the "Suns2" system. The following provides the steps to compile and execute the DDB.

- Obtain a fresh copy of the Vbase Kernel Database.
- Change directories to the "tdl" directory.
- Type "tdl -v *.tdl" to compile the tdl definitions.
- Change directories to the "methods" directory.
- Type "make application_program_name". Application_program_name is the name of the application program to compile, i.e. createRootNode. This command will compile the COP operations code as well as the application program. Type this command until all applications programs are compiled.
- Type "application program name" to invoke the application.

The make file for the Design Database is as follows.

```
CEFLAGS=      -g
CLFLAGS =
CFLAGS =      $(CEFLAGS) $(CLFLAGS)

.c.o:;     cop -c $(CFLAGS) $*.c


LIBRARY =         -lvbase -lm -ll
SOURCES       =      Node.c

OBJECTS =     $(SOURCES:.c=.o)
```

```
traverseTree: $(OBJECTS) traverseTree.o
    cop $(CFLAGS) -o traverseTree      \
    $(OBJECTS) traverseTree.o     -lvbase -lm -ll

createRootNode: $(OBJECTS) createRootNode.o
    cop $(CFLAGS) -o createRootNode  \
    $(OBJECTS) createRootNode.o   -lvbase -lm -ll


createChildNode: $(OBJECTS) createChildNode.o
    cop $(CFLAGS) -o createChildNode \
    $(OBJECTS) createChildNode.o  -lvbase -lm -ll

storeProperty: $(OBJECTS) storeProperty.o
    cop $(CFLAGS) -o storeProperty  \
    $(OBJECTS) storeProperty.o    -lvbase -lm -ll

getProperty: $(OBJECTS) getProperty.o
    cop $(CFLAGS) -o getProperty       \
    $(OBJECTS) getProperty.o      -lvbase -lm -ll

getParent: $(OBJECTS) getParent.o
    cop $(CFLAGS) -o getParent \
    $(OBJECTS) getParent.o     -lvbase -lm -ll

getChildren: $(OBJECTS) getChildren.o
    cop $(CFLAGS) -o getChildren \
    $(OBJECTS) getChildren.o   -lvbase -lm -ll

deleteNode: $(OBJECTS) deleteNode.o
    cop $(CFLAGS) -o deleteNode \
    $(OBJECTS) deleteNode.o    -lvbase -lm -ll

CLEAN  :
      rm -f *.o a.out core traverseTree createRootNode\
      createChildNode storeProperty getProperty       \
      getParent getChildren deleteNode
```

LIST OF REFERENCES

1. Luqi and Ketabchi, M., *A Computer Aided Prototyping System*, Tech. Rep. NPS 52-87-011, Naval Postgraduate School, Monterey, CA, 1987 and in IEEE Software, pp. 66-72, March 1988.

2. Boehm, B.W., "Improving Software Productivity," *IEEE Computer*, pp.43-57, September 1988.

3. Booch, G., *Software Engineering with Ada*, Benjamin Cummings Publishing Co., Inc., Menlo Park, CA, 1983.

4. Chitwood, G., "Ada Meets the Challenge of Real-Time Simulation," *Defense Computing*, v. 1, no. 4, pp.32-38, July/August 1988.

5. Nielsen, K., and Shumate, K., *Designing Large Real-Time Systems With Ada*, Intertext Publications/Multiscience Press, Inc., New York, NY, 1988.

6. Galik, D., *A Conceptual Design of a Software Base Management System for the Computer Aided Prototyping System*, Master's thesis, Naval Postgraduate School, Monterey, CA, December 1988.

7. Raum, H., *Design and Implementation of an Expert User Interface for the Computer Aided Prototyping System*, Master's thesis, Naval Postgraduate School, Monterey, CA, December 1988.

8. Altizer, C., *Implementation of a Language Translator for the Computer Aided Prototyping System*, Master's thesis, Naval Postgraduate School, Monterey, CA, December 1988.

9. Thorstenson, R., *A Graphical Editor for the Computer Aided Prototyping System*, Master's thesis, Naval Postgraduate School, Monterey, CA, December 1988.

10. Wood, M., *Run-Time Support for Rapid Prototyping*, Master's thesis, Naval Postgraduate School, Monterey, CA, December 1988.

11. Porter, S., *Design of a Syntax Directed Editor for PSDL*, Masters thesis, Naval Postgraduate School, Monterey, CA, December 1988.

12. Marlowe, L., *A Scheduler for Critical Time Constraints*, Masters thesis, Naval Postgraduate School, Monterey, CA, December 1988.

13. Ketabchi, M., "The Object-Oriented Model," *Proceedings of the 1986 Fall Joint Computer Conference*, Computer Society Press of the IEEE, Washington, D.C., 1987.

14. McKenna, J., "Teaching OOP," *OOPSLA '88 Conference Proceedings*, The Association for Computing Machinery, New York, NY, 1988.

15. Ketabchi, M., and Berzins, V., "Modeling and Managing CAD Databases," *IEEE Computer*, pp.46-49, February 1987.

16. Ketabchi, M., *Object Oriented Database Management Systems for Complex Data and Process Intensive Applications* (Course Notes), September 1988.

17. *Vbase Integrated Object Database User's Manual*, Ontologic Inc., Billerica, MA, 1987.

18. Berzins, V., and Luqi, *An Introduction to the Specification Language SPEC*, Tech. Rep. NPS 52-88-031, Naval Postgraduate School, Monterey, CA, 1988.

19. Berzins, V., *Software System Design* (Course Notes), 1988.

20. Landis, G., "Maintaining Design Evolution and History Information in an Object-Oriented CAD/CAM Database", Ontologic, Inc., 1987.

INITIAL DISTRIBUTION LIST

1.	Defense Technical Information Center Cameron Station Alexandria, Virginia 22304-6145	2
2.	Library, Code 0142 Naval Postgraduate School Monterey, California 93943-5000	2
3.	Office of Naval Research Office of the Chief of Naval Research Attn. CDR Michael Gehl, Code 1224 800 N. Quincy Street Arlington, Virginia 22217-5000	1
4.	Space and Naval Warfare Systems Command Attn. Dr. Knudsen, Code PD 50 Washington, D.C. 20363-5100	1
5.	Ada Joint Program Office OUSDRE(R&AT) Pentagon Washington, D.C. 20301	1
6.	Naval Sea Systems Command Attn. CAPT Joel Crandall National Center #2, Suite 7N06 Washington, D.C. 20363-5100	1
7.	Office of the Secretary of Defense Attn. CDR Barber STARS Program Office Washington, D.C. 20301	1
8.	Office of the Secretary of Defense Attn. Mr. Joel Trimble STARS Program Office Washington, D.C. 20301	1
9.	Commanding Officer Naval Research Laboratory Code 5150 Attn. Dr. Elizabeth Wald Washington, D.C. 20375-5000	1

10.	Navy Ocean System Center Attn. Linwood Sutton, Code 423 San Diego, California 92152-5000	1
11.	National Science Foundation Attn. Dr. William Wulf Washington, D.C. 20550	1
12.	National Science Foundation Division of Computer and Computation Research Attn. Dr. Peter Freeman Washington, D.C. 20550	1
13.	National Science Foundation Director, PYI Program Attn. Dr. C. Tan Washington, D.C. 20550	1
14.	Office of Naval Research Computer Science Division, Code 1133 Attn. Dr. Van Tilborg 800 N. Quincy Street Arlington, Virginia 22217-5000	1
15.	Office of Naval Research Applied Mathematics and Computer Science, Code 1211 Attn. Mr. J. Smith 800 N. Quincy Street Arlington, Virginia 22217-5000	1
16.	Defense Advanced Research Projects Agency (DARPA) Integrated Strategic Technology Office (ITSO) Attn. Dr. Jacob Schwartz 1400 Wilson Boulevard Arlington, Virginia 22209-2308	1
17.	Defense Advanced Research Projects Agency (DARPA) Integrated Strategic Technology Office (ITSO) Attn. Dr. Squires 1400 Wilson Boulevard Arlington, Virginia 22209-2308	1
18.	Defense Advanced Research Projects Agency (DARPA) Integrated Strategic Technology Office (ITSO) Attn. MAJ Mark Pullen, USAF 1400 Wilson Boulevard Arlington, Virginia 22209-2308	1

19. Defense Advanced Research Projects Agency (DARPA) 1
Director Naval Technology Office
1400 Wilson Boulevard
Arlington, Virginia 22209-2308

20. Defense Advanced Research Projects Agency (DARPA) 1
Director, Strategic Technology Office
1400 Wilson Boulevard
Arlington, Virginia 22209-2308

21. Defense Advanced Research Projects Agency (DARPA) 1
Director, Prototype Projects Office
1400 Wilson Boulevard
Arlington, Virginia 22209-2308

22. Defense Advanced Research Projects Agency (DARPA) 1
Director, Tactical Technology Office
1400 Wilson Boulevard
Arlington, Virginia 22209-2308

23. COL C. Cox, USAF 1
JCS (J-8)
Nuclear Force Analysis Division
Pentagon
Washington, D.C. 20318-8000

24. LTCOL Kirk Lewis, USA 1
JCS (J-8)
Nuclear Force Analysis Division
Pentagon
Washington, D.C. 20318-8000

25. U.S. Air Force Systems Command 1
Rome Air Development Center
RADC/COE
Attn. Mr. Samuel A. DiNitto, Jr.
Griffis Air Force Base, New York 13441-5700

26. U.S. Air Force Systems Command 1
Rome Air Development Center
RADC/COE
Attn. Mr. William E. Rzepka
Griffis Air Force Base, New York 13441-5700

27. Professor Valdis Berzins 1
Code 52BE
Naval Postgraduate School
Computer Science Department
Monterey, California 93943-5000

28. LT Bryant S. Douglas 1
1217 N. Clay
Springfield, Missouri 65802